Energy-Efficient Run-Time Mapping and Thread Partitioning of Concurrent OpenCL Applications on CPU-GPU MPSoCs

AMIT KUMAR SINGH, University of Southampton
ALOK PRAKASH, Nanyang Technological University
KARUNAKAR REDDY BASIREDDY, GEOFF V. MERRETT, and BASHIR M. AL-HASHIMI, University of Southampton

Heterogeneous Multi-Processor Systems-on-Chips (MPSoCs) containing CPU and GPU cores are typically required to execute applications concurrently. However, as will be shown in this paper, existing approaches are not well suited for concurrent applications as they are developed either by considering only a single application or they do not exploit both CPU and GPU cores at the same time. In this paper, we propose an energy-efficient run-time mapping and thread partitioning approach for executing concurrent OpenCL applications on both GPU and GPU cores while satisfying performance requirements. Depending upon the performance requirements, for each concurrently executing application, the mapping process finds the appropriate number of CPU cores and operating frequencies of CPU and GPU cores, and the partitioning process identifies an efficient partitioning of the applications' threads between CPU and GPU cores. We validate the proposed approach experimentally on the Odroid-XU3 hardware platform with various mixes of applications from the Polybench benchmark suite. Additionally, a case-study is performed with a real-world application SLAMBench. Results show an average energy saving of 32% compared to existing approaches while still satisfying the performance requirements.

CCS Concepts: • **Computer systems organization → Embedded systems**;

Additional Key Words and Phrases: Heterogeneous MPSoC, OpenCL applications, Run-time management, Performance, Energy consumption

ACM Reference format:
Amit Kumar Singh, Alok Prakash, Karunakar Reddy Basireddy, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2017. Energy-Efficient Run-Time Mapping and Thread Partitioning of Concurrent OpenCL Applications on CPU-GPU MPSoCs. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 147 (September 2017), 22 pages.
https://doi.org/10.1145/3126548

"This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue".
This work was supported in parts by the Engineering and Physical Sciences Research Council (EPSRC) Programme Grant EP/L000563/1 and the PRiME Programme Grant EP/K034448/1 (www.prime-project.org).
Authors' addresses: A. K. Singh, K. R. Basireddy, G. V. Merrett, B. M. Al-Hashimi, School of Electronics and Computer Science, University of Southampton, United Kingdom SO17 1BJ; A. Prakash, School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798.

https://doi.org/10.1145/3126548

1 INTRODUCTION

Modern embedded systems, e.g. mobile phones, rely on heterogeneous Multi-Processor Systems-on-Chips (MPSoCs) containing different types of cores. An example of a commercial heterogeneous MPSoC is the Samsung Exynos 5422 SoC [5], which powers the popular Samsung Galaxy series of mobile phones. This SoC contains 4 ARM Cortex-A15 (big) CPU, 4 ARM Cortex-A7 (LITTLE) CPU and 6 ARM Mali-T628 GPU cores. Such an architecture provides opportunities to exploit distinct features of different types of cores in order to meet end-user demands in terms of performance and energy consumption. Additionally, the cores in such MPSoCs support dynamic voltage and frequency scaling (DVFS) that can be exploited to reduce dynamic power consumption ($P \propto V^2 f$). By employing DVFS, the energy consumption can be minimized if the power consumption is reduced sufficiently to account for the extra time taken to run the workload at a lower voltage and frequency. In some situations, dynamic power management (DPM) can lead to lower energy consumption than DVFS [20, 23].

For a given application, simultaneous exploitation of heterogeneous cores having different instruction set architectures (ISAs) such as CPU and GPU is challenging, as they handle instructions in different ways. Additionally, CPU cores typically handle task and thread level parallelisms, whereas GPU cores handle data level parallelism. OpenCL [7] provides an opportunity to write programs that can execute across heterogeneous cores including CPUs and GPUs [18, 19, 24, 30]. However, depending upon the kind of parallelism dominant in the application, the performance and energy consumption will vary when it is allocated onto only CPU, only GPU, or both CPU and GPU cores.

Figure 1 shows execution time (bars) and energy consumption (lines) when executing individual OpenCL applications (SYR2K, SYRK, CORRELATION (CORR), and COVARIANCE (COVR)) from the Polybench benchmark [16] on the Exynos 5422 heterogeneous MPSoC while varying the fraction of application workload (threads) executed on CPU cores and remaining threads on GPU cores. All the cores are set to operate at maximum possible voltage-frequency. A fraction value of zero indicates that no threads are executed on CPU cores, i.e. all of them are executed only on the GPU cores. Similarly, when this value is 1, all the threads are executed only on CPU cores and none on GPU cores. It can be observed that some applications (specially having substantial sequential fraction) execute faster on CPU cores than GPU cores (e.g., CORR), whereas others (having extensive data parallelism) finish early on GPU cores (e.g., COVR). Further, all applications show a significant reduction in execution time and energy consumption when run on both the CPU and GPU cores, with an appropriate fraction value (between 0 and 1), defining the best partitioning of threads between CPU and GPU cores. These observations clearly indicate the advantages of simultaneously exploiting both CPU and GPU cores for each application.

It is well known that multiple applications executing concurrently in a modern embedded system need to satisfy their performance requirements, and the overall energy consumption should be optimized [33]. For example, a mobile phone might need to execute JPEG and MP3 decoding concurrently, while satisfying the respective frames per second (fps) requirements when a user is viewing images and listening to music at the same time. However, existing run-time management works for concurrent applications consider only single-ISA heterogeneous MPSoCs, where cores have the same instruction set architecture [9, 10, 12, 13, 25, 34]. They usually consider Pthreads programming model and cannot be used to simultaneously exploit cores of different ISAs such as CPU and GPU. For a single application, simultaneous exploitation of CPU and GPU cores has been performed by employing OpenCL programming model [30], but it leads to poor results when applied to concurrent applications (shown in the next section). Additionally, some works exploit either CPU or GPU for an application [37], which is not efficient in terms of execution time and

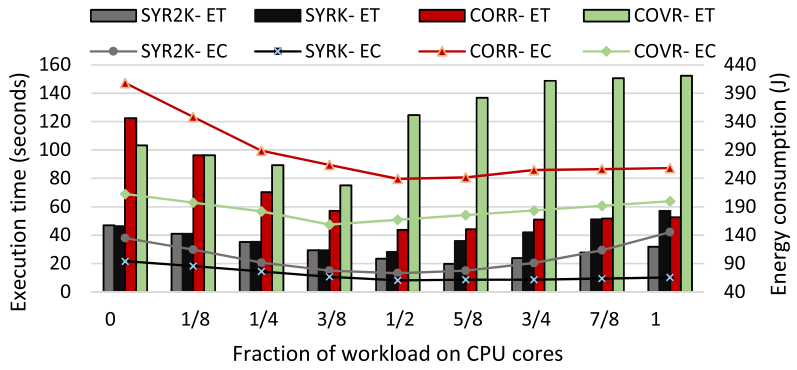

Fig. 1. Execution time (ET) and energy consumption (EC) at varying fraction of application workload (threads) to be executed on CPU cores.

energy consumption as shown in Figure 1. These observations indicate that existing run-time management approaches lack the ability to efficiently exploit both CPU and GPU cores of a MPSoC at the same time for concurrent applications. This necessitates designers to address the challenge of identifying appropriate mapping of application threads to cores and the partitioning of threads between CPU and/or GPU cores to satisfy the performance requirements and optimize overall energy consumption. Since mapping and partitioning determine the execution time and energy consumption, it is important to find them appropriately. The mapping is determined by the number of used CPU cores and operating frequencies of CPU and GPU cores, and partitioning needs to be determined by taking applications' stretching (extended execution time) due to co-scheduling and CPU/GPU cores processing capability into account. Since several factors control mapping and partitioning, it is challenging to identify the best mapping and partitioning for each application.

This paper addresses the aforementioned mapping and partitioning challenges by making the following concrete contributions.

(1) For executing multiple performance constrained OpenCL applications concurrently, a run-time management approach that performs energy efficient mapping and repartitioning of threads of each application between CPU and GPU cores of a MPSoC while taking the applications' stretching (extended execution time) due to co-scheduling into account.
(2) To enable efficient run-time mapping and repartitioning, a design-time strategy to identify best mapping and threads partitioning options for each application. These options are stored as design points that are used to identify mapping and repartitioning of concurrent applications by taking applications' stretching into account.
(3) Implementation of the offline and run-time steps on a real hardware platform, specifically Odroid-XU3 platform [6], which contains state-of-the-art mobile MPSoC (Samsung Exynos 5422) prevalent in smartphone/tablet devices.

To the best of our knowledge, this is the first study on energy efficient run-time mapping and partitioning of threads of concurrent applications on CPU and GPU cores of a heterogeneous MPSoC while satisfying the performance requirements.

The remainder of this paper is organized as follows. Section 2 provides a motivational case study by considering some run-time concurrent application scenarios. Section 3 presents related works. System and problem definition are introduced in Section 4. Section 5 describes various stages of the

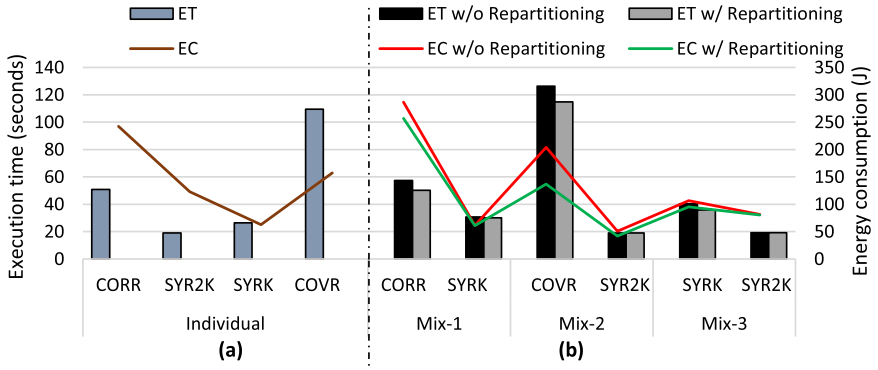

Fig. 2. Execution time (ET) and total energy consumption (EC) for executing (a) individual and (b) concurrent applications (Mix) without (w/o) and with (w/) repartitioning.

proposed run-time management methodology. Section 6 presents the experimental results. Finally, Section 7 concludes the paper.

2 MOTIVATIONAL CASE STUDY

We first present a case study to illustrate the shortcomings of current approaches and potential of the proposed approach. Figure 2(a) shows execution time (bar) and total energy consumption (line) when applications are executed individually by using the best individual applications' threads partitioning leading to minimum energy consumption, which can be identified from Figure 1. Here, when using CPU cores, applications CORR and SYR2K are mapped on 4 big cores each, and SYRK and COVR on 4 LITTLE cores each. This is possible as Samsung Exynos 5422 MPSoC tool chains allow application threads to be allocated to a subset of CPU cores. By not using all the CPU cores for an application, spatially isolated allocation of cores to various applications can be performed. Such spatially isolated execution of applications on the CPU cores results in high predictability and helps to decide appropriate number of CPU cores and their types to be allocated for satisfying the performance requirements. However, GPU driver doesn't support the spatially isolated and time multiplexed execution of multiple applications. Even the latest GPUs [1, 3, 4] in mobile processors do not support such execution due unavailability of appropriate drivers. Therefore, each application uses all GPU cores till its completion. This implies that in case of multiple applications to be executed concurrently, spatially isolated or time multiplexed execution is not possible on the GPU side and it will lead to stretched overall execution.

Figure 2(b) shows execution time of individual applications and total energy consumption when applications are executed in various combinations (Mix-1, Mix-2 and Mix-3) by using the best individual applications' threads partitioning referred to as without (w/o) repartitioning and by using the partitioning obtained by our repartitioning approach referred to as with (w/) repartitioning. For w/o repartitioning, the partition is the same as in Figure 2(a), which does not consider applications' stretching (extended execution time) due to co-scheduling on GPU cores, e.g. in [30]. In contrast, applications' stretched executions are considered in w/ repartitioning. In w/o repartitioning, for each combination of applications (Mix), it can be observed that execution time of one application is stretched (e.g., COVR in Mix-2) as compared to individual execution (e.g., COVR in Figure 2(a)) because sequential en-queuing of threads takes place on the GPU side. The stretched execution leads to increased energy consumption as well. In w/ repartitioning, applications' stretching (specially on the GPU side as execution cannot be spatially isolated or time multiplexed) has been

taken into account to repartition individual applications' threads between CPU and GPU cores such that balanced execution can be performed. For each application, a balanced execution implies that completion on chosen CPU cores and GPU cores takes place almost at the same time, which also avoids waiting of threads completion on one type of core. This has led to reduced execution time that is determined as the maximum of the time taken on CPU and GPU cores. To achieve balanced execution, the repartitioning approach moves higher fraction of threads to CPU side as GPU side is performing sequential en-queuing and execution of applications' threads, i.e. one application after another. It can be observed that repartitioning has also led to reduced energy consumption due to reduction in execution time.

3 RELATED WORKS

Run-time mapping of multi-threaded applications on single-ISA heterogeneous MPSoCs has been a hot topic [9, 10, 12, 13, 25, 34]. Most of these approaches consider Samsung Exynos 5422 SoC and utilize 4 big and/or 4 LITTLE cores that have the same ISA [9, 12, 13]. Further, for a given application, most of these approaches do not concurrently exploit more than one types of cores [12, 13, 25, 34]. Although there has been some effort to concurrently exploit both big and LITTLE cores [9], it cannot be applied to exploit cores having different ISAs such as CPU and GPU because they handle instructions in different ways.

There has been efforts to simultaneously exploit CPU and GPU cores in desktop platforms, but CPU and GPU cores are not situated within a single chip [18, 19, 24, 27, 37]. In these works, CPU cores are used for general purpose tasks and GPU cores to accelerate data-parallel tasks. Such allocation of tasks to cores leads to improved throughput and energy efficiency. However, most of these approaches perform static mapping for an application [18, 19] and thus they cannot be applied to perform run-time mapping and partitioning of threads of multiple applications. Some approaches consider multiple applications [24], but the applications are dispatched either to CPU cores or GPU cores. Further, since CPU and GPU cores are situated in different chips, these approaches cannot be efficiently applied to MPSoC due to different communication infrastructure.

For desktop platforms, there has also been efforts to exploit CPU and GPU cores present within a single chip [28, 35, 36]. In these platforms, coordination of CPU and GPU cores needs more consideration. In [35], a run-time algorithm is proposed to partition the workload and power budget between CPU and GPU cores of an AMD Trinity single chip heterogeneous platform to improve throughput. In [36], similar AMD platform is used to perform coordinated CPU-GPU executions, but memory contention occurs due to access of the same bank in different patterns by the CPU and GPU. In [28], the problem of shared resources in AMD platforms is addressed. However, these efforts do not consider limited power budget that is available for embedded systems operating from batteries.

For mobile platforms used in embedded systems and containing CPU and GPU cores within a single chip, there has been some works to partition the application threads between CPU and GPU cores. In [15], HPC workloads are executed on Mali GPU to achieve energy efficiency, but the possible collaboration with CPU is not considered. In [11], the threads are partitioned by considering shared resources and synchronization. However, these works do not use GPU for OpenCL kernel execution. OpenCL framework for ARM processors was introduced in [21]. In [30], a similar open source framework, FreeOCL [8] is used for the ARM CPU that acts as both the host processor and an OpenCL device. This enables concurrent use of CPU and GPU to execute an application threads, but in [30], a static partitioning is performed by using all the CPU and GPU cores.

A close observation of approaches to map and partition application threads between CPU and GPU cores of a mobile MPSoC indicates that they cannot be efficiently applied for run-time mapping and partitioning of the threads of concurrent applications. Further, while doing such

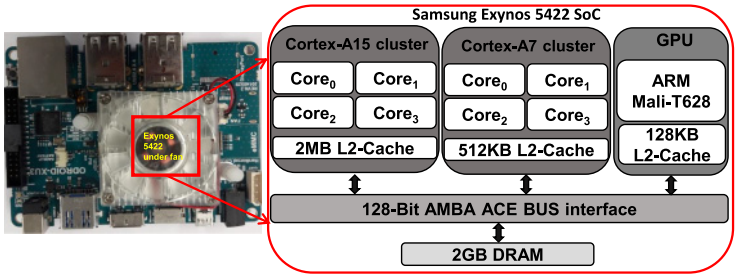

Fig. 3. Odroid-XU3 board containing Samsung Exynos 5422 heterogeneous MPSoC.

partitioning, existing approaches do not consider performance constrained applications. In contrast, our proposed approach performs energy-efficient mapping and partitioning of applications' threads while respecting the performance constraints of each application.

4 SYSTEM AND PROBLEM FORMULATION

This section presents hardware and software infrastructure to represent the system and the detailed problem formulation.

4.1 Heterogeneous MPSoC

Modern heterogeneous MPSoCs contain different types of cores having the same or different ISAs and the number of cores of each type can vary. Usually, the cores of the same type are situated with a cluster. We consider a similar heterogeneous MPSoC in our work. In particular, we consider Samsung Exynos 5422 MPSoC present on the Odroid XU3 board [6]. Figure 3 shows the Odroid-XU3 board and more insight of the Exynos 5422 MPSoC. This MPSoC is based on ARM's big.LITTLE technology [2] and contains a cluster of 4 ARM Cortex-A15 (big) CPU cores and another of 4 ARM Cortex-A7 (LITTLE) CPU cores. These cores implement ARM v7A ISA. Each core has private L1 instruction and data cache and L2 is shared across all the cores within a cluster. Additionally, it also contains 6 ARM Mali-T628 GPU shader cores based on "Midgard" architecture and 2GB DRAM LPDDR3. The main component of a shader core is a programmable massively multi-threaded "tri pipe" processing engine that contains one load-store pipeline, two arithmetic pipelines, and one texture pipeline.

This MPSoC also provides DVFS feature per cluster. For Cortex-A15 cluster, the frequency can be varied between 200 MHz to 2000MHz with a 100 MHz step, whereas for Cortex-A7 cluster, it can be varied between 200 MHz to 1400 MHz with a step of 100 MHz. The frequency of GPU cluster can be set at 177 MHz, 266 MHz, 350 MHz, 420 MHz, 480 MHz, 543 MHz and 600 MHz. It should be noted that we vary only frequency, but firmware automatically adjusts the voltage based on pre-set pairs of voltage-frequency values.

4.2 Software Infrastructure

4.2.1 Operating System Support. The Odroid XU3 board supports different flavours of Linux. In particular, we use popular Ubuntu 14.04 LTS. This version supports Heterogeneous Multi-Processing (HMP) that enables use of all the CPU cores (big and LITTLE) simultaneously. Additionally, it supports DVFS of different cluster cores by editing the appropriate virtual files for the corresponding devices in the Linux *sysfs* directory. It also supports core disabling of CPU cores that provides opportunity to use selective big and/or LITTLE cores to execute an application.

4.2.2 OpenCL. The Open Computing Language (OpenCL) [7] is an open standard for developing parallel applications to exploit heterogeneous multi-core architectures [22, 38]. OpenCL vendors provide runtime software and compilation tools to facilitate execution of OpenCL programs (applications) on supported devices, e.g. CPU and GPU. It also supports exploitation of multiple devices by a single program.

In OpenCL computation model, a computing system consists of a number of devices attached to a host processor that is usually a CPU. The host acts like a manager and controls compute devices for performing computations referred to as kernels. A compute device could be a CPU, GPU, or DSP. The host sets up devices, create kernels, build them and finally send them to devices for execution by using OpenCL APIs. It also sends/receives data to/from devices before and after the execution. Each compute device (e.g., GPU) consists of compute units (e.g., shader cores) and each compute unit consists of processing elements (e.g., arithmetic pipelines). A processing element executes a kernel instance called as work-item that operates on a single data point. A group of work-items form a work-group and these items execute concurrently on the processing elements of a single compute unit. The data processed by OpenCL is in an index space of work-items that are organized in an N-Dimensional Range (NDRange). The OpenCL memory model demands memory consistency across work-items within a work-group but not among work-groups. Therefore, without worrying about maintaining memory consistency among compute devices, different work-groups can be launched on different devices (e.g., CPU and GPU).

4.2.3 FreeOCL. FreeOCL [8] is an open-source OpenCL runtime library that provides OpenCL support for the ARM CPU cores. In [21], a similar OpenCL framework is described to support ARM processors. Such support is typically not available in current mobile SoCs as OpenCL runtime software is supplied only for the Mali GPU to promote its usage for general purpose computing or acceleration. The compilation and installation of FreeOCL enabled ARM CPU to act as both host processor and an OpenCL device. Thus, both CPU and GPU cores can be concurrently exploited for executing an application. This provides opportunity to partition an application threads between CPU and GPU cores towards fast completion of the application.

4.2.4 Applications. The data-parallel applications are potential candidates to concurrently exploit cores of a MPSoC as data can be processed in parallel on the cores. However, each application should be written in OpenCL to exploit cores of two different ISAs such as CPU and GPU. The GPU version of the popular Polybench benchmark suite [16] contains such data-parallel applications written in OpenCL and we use them. The application codes are slightly modified to launch them only on CPU cores, only on GPU cores, or on both CPU and GPU cores. Additionally, an appropriate work-group size for each application is selected as in [30]. For each application, the user can specify a performance requirement in terms of completion time of the application. This timing requirement can be translated to throughput requirement for frame based application like audio/video processing, where throughput is expressed as a frame rate to guarantee a good user experience.

4.3 Problem Definition

For performance constrained data-parallel applications to be executed on a GPU-GPU heterogeneous MPSoC, several thread-to-core mapping options exist for each application as its threads can be partitioned between CPU and GPU cores while using different combinations of big/LITTLE CPU cores and/or the GPU cores. For n_b big and n_L LITTLE CPU cores, the total number of mappings (M_{CPU}) is:

$$M_{CPU} = n_b + n_L + (n_b \times n_L) \tag{1}$$

For GPU cores, there will be only one mapping as all the cores will be used.

$$M_{GPU} = 1 \tag{2}$$

Since the considered MPSoC supports cluster-wide DVFS, the cores within each cluster can be set to a voltage-frequency level from a predefined set of voltage-frequency pairs [17]. Let, F_b, F_L and F_g be the number of voltage-frequency levels for big, LITTLE and GPU cluster, respectively, the mapping design space considering voltage-frequency levels for the CPU ($M_{CPU_{VF}}$) and GPU ($M_{GPU_{VF}}$) cores will be as follows.

$$M_{CPU_{VF}} = (n_b \times F_b) + (n_L \times F_L) + (n_b \times F_b \times n_L \times F_L) \tag{3}$$

$$M_{GPU_{VF}} = 1 \times F_g \tag{4}$$

The total number of combined design points (CDP) considering both the CPU and GPU cores are:

$$\begin{aligned} CDP &= M_{CPU_{VF}} \times M_{GPU_{VF}} \\ &= \{(n_b \times F_b) + (n_L \times F_L) + (n_b \times F_b \times n_L \times F_L)\} \times (1 \times F_g) \end{aligned} \tag{5}$$

For a given application, the fraction of work-items (threads) to be executed on the CPU and GPU cores can be identified by utilizing the above design space. Let us consider a total of T work-items and N fraction of work-items to run on CPU cores and rest on GPU cores at a particular voltage-frequency setting. The execution time (ET) of the application after splitting between CPU and GPU cores can be estimated as follows.

$$ET = max\{N \times ET_{CPU}, (T - N) \times ET_{GPU}\} \tag{6}$$

Where, ET_{CPU} and ET_{GPU} are estimated execution time for CPU-only and GPU-only executions. This equation indicates that execution time will be determined by the device taking more time to execute its assigned work-items.

The energy consumption (EC) can be estimated as:

$$EC = EC_{CPU} + EC_{GPU} + EC_{MEM} \tag{7}$$

Where, EC_{CPU}, EC_{GPU} and EC_{MEM} are the energy consumptions of CPU cores, GPU cores and memory, respectively, which can be computed at the product of respective power consumption and execution time.

Since we have several mapping options (design points) on both CPU (Equation 3) and GPU (Equation 4) side, we will have different values of ET_{CPU} and ET_{GPU} for those points. Similarly, Equation (7) will lead to different values of energy consumption. This indicates that an appropriate partition needs to be identified for each combination of design points from the CPU and GPU side. However, as such number of combinations are going to be large, the partitioning considering each combination is time consuming and thus cannot be done at run-time. Therefore, it can be shifted to design-time and the partitioning results can be used to facilitate energy efficient run-time mapping and partitioning of work-items of multiple OpenCL applications while satisfying the performance requirements. This defines the problem as follows.

Given an application or a set of concurrent applications with performance constraints and a heterogeneous CPU-GPU MPSoC supporting DVFS

Find energy efficient partitioning of work-items of each application between CPU and GPU cores along with thread-to-core mapping

subject to meeting performance requirement of each application and not exceeding available MPSoC resources

At run-time, for each application, a new partitioning needs to be identified to optimize energy consumption as the design-time partitioning will not lead to efficient results due to applications

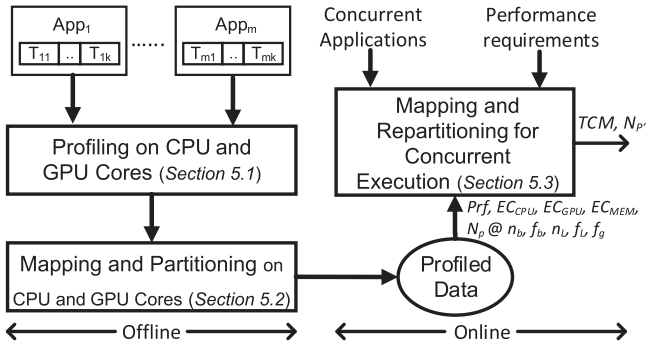

Fig. 4. Proposed thread mapping and repartitioning approach.

stretched execution time (shown earlier in Figure 2). The energy efficient partitioning refers to identify the partitions such that overall energy consumption will be optimized and it needs to be identified by considering the design points of concurrent applications.

5 PROPOSED RUN-TIME THREAD MAPPING AND PARTITIONING

An overview of the proposed run-time thread mapping and partitioning approach is illustrated in Figure 4. The approach has some offline and online steps (falling into the category of hybrid approach [31]) where offline computed results for various applications are used to identify energy efficient run-time mapping and partitioning of threads of concurrent applications to be executed on the heterogeneous MPSoC. The main steps of the approach are as follows:

(1) Offline profiling on CPU and GPU cores (Section 5.1).
(2) Offline mapping and partitioning of threads on CPU and GPU cores (Section 5.2).
(3) Online (run-time) mapping and repartitioning for concurrent execution of applications (Section 5.3).

The novel aspects of our run-time mapping and repartitioning approach are as follows.

- Run-time identification of applications' stretching (extended execution time) due to their concurrent execution.
- Consideration of CPU and GPU cores processing capability and applications' stretched execution to identify the repartitioned work-groups.
- Accurate energy consumption estimation to evaluate the mapping and repartitioning options in order to select the best one.
- Execution time estimation of applications for different mapping and repartitioning options to verify against timing requirement.

The following sections provide more details of each step.

5.1 Offline Profiling on CPU and GPU Cores

For each available application (App_1 to App_m), the profiling step for CPU cores computes all the possible mappings and their execution time when using various combinations of big and LITTLE CPU cores operating at different frequencies. On CPU cores, the total number of mappings for each application can be computed by following Equation (3). For the considered MPSoC, the total number of CPU mappings (design points) for each application is 4080 ($M_{CPU_{VF}}$ = 4×19 + 4×13 + 4×19 × 4×13) based on the fact that there are 4 big and 4 LITTLE CPU cores that can operate at

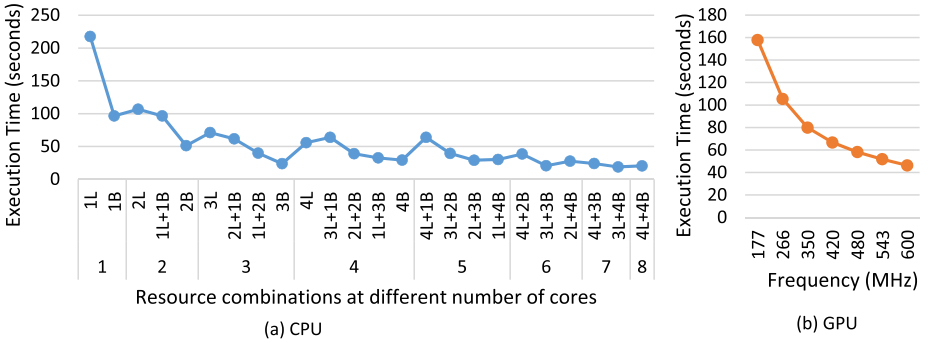

Fig. 5. Profiling results on CPU and GPU cores.

19 and 13 frequency levels, respectively. To illustrate some of the design points, Figure 5(a) shows part of the profiling results in terms of execution time for SYR2K application from the Polybench benchmark when the big and LITTLE CPU cores operate at the respective maximum frequency and are used in various combinations (on the horizontal axis), i.e. only 24 (4×1 + 4×1 + 4×1 × 4×1) points.

For GPU cores, the profiling step computes all the possible design points and their performance when the GPU cores operate at different frequencies. The total number of GPU design points for each application is 7 ($M_{GPU_{VF}}$ = 1×7, from Equation (4)) based on the fact the application threads are mapped on all GPU cores and 7 frequency levels are available for the GPU cluster. Figure 5(b) shows the GPU profiling results for the SYR2K application.

5.2 Offline Mapping and Partitioning of Threads Between CPU and GPU Cores

For each application, this step determines the fraction of work-items to be executed on CPU and GPU cores by considering the design points obtained in the previous step. For all the possible combination of design points between CPU and GPU, one appropriate partitioning of work-items between CPU and GPU leading to optimized execution time is possible. Such a partitioning should lead to the completion of work-items on the CPU and GPU side at the same time. For facilitating application execution as OpenCL kernels, the workload on both CPU and GPU should be multiples of work-group size. Therefore, the partitioning point can be calculated as the number of work-groups that is nearest to the desired fraction of the CPU workload.

The individual capacities of CPU and GPU design points can be considered to accomplish the desired partitioning [30]. The individual capacity can be measured in terms of execution time of the design point. Let W be the total number of work-groups in an application and ET_{CPU_p} and ET_{GPU_p} are the execution time on CPU and GPU for the chosen combination of design point (p) from CPU and GPU, then the fraction N_p of the work-groups that should be executed on the CPU can be computed by assuming N_p work-groups on CPU and $(W - N_p)$ work-groups on GPU will complete the execution at the same time, i.e.:

$$N_p \times ET_{CPU_p} = (W - N_p) \times ET_{GPU_p} \tag{8}$$

Equation (8) can be derived as:

$$N_p = \frac{W}{1 + \frac{ET_{CPU_p}}{ET_{GPU_p}}} \tag{9}$$

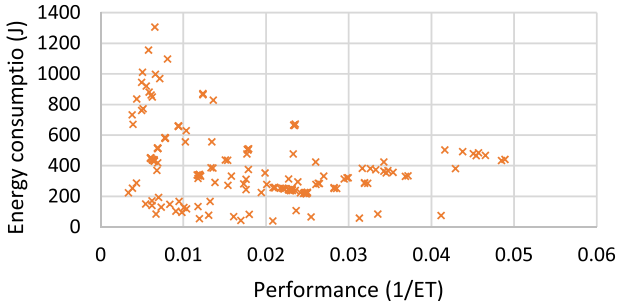

Fig. 6. Design points representing performance and energy consumption for SYR2K application.

Such partitioning leads to load balancing of work-items between CPU and GPU cores. For each combination of design points from CPU and GPU, we can obtain the best partitioning by utilizing Equation (9) while considering execution times on the CPU and GPU cores. For the considered MPSoC, the number of combination (design) points for each application is 28560 (=4080 × 7) considering 4080 and 7 design points on the CPU and GPU cores, respectively, and can be computed by Equation (5).

Each of the combination point give a new design point (D_p) that can be represented in terms of partition (N_p), number of used big cores (n_b) and their frequency ($f_b \in F_b$), number of used LITTLE cores (n_L) and their frequency ($f_L \in F_L$) and frequency of GPU cores (f_g).

For each design point D_p, the application is again executed to simultaneously exploit CPU and GPU cores by following partition N_p. The application execution time and energy consumption are computed by following Equation (6) and (7), respectively. Thus, we obtain performance (1/Execution-time) and energy consumption at each design point D_p. Similar computations are performed for all the applications.

Considering all the terms, each design point can be represented by a 10-tuple as:

$D_p = (N_p, n_b, f_b, n_L, f_L, f_g, Prf, EC_{CPU}, EC_{GPU}, EC_{MEM})$.

Figure 6 shows example design points (168 = 24×7) for the SYR2K application corresponding to the individual profiling on CPU and GPU cores shown in Figure 5. However, there are a total of 28560 points as mentioned earlier. In order to store only efficient points in terms of performance, energy consumption and resource usage, we distil the points as follows. First, out of all the design points (28560), we consider performance and energy efficient points at various possible CPU and GPU cores combinations. Then, if performance of a point using higher number of cores is the same or smaller than the performance of a point using lower number of cores, energy consumption in latter point is the same or lower than the former point and cores in the latter point are a subset of cores in the former point, then the former point is discarded. This results in Pareto-optimal points, where each such point is better than another in terms of performance, energy consumption or resource usage. The design points can be distilled by using other techniques as well [29]. These distilled points (combination points) for each application are stored after sorting them in descending order based on the performance Prf. Such storing helps to easily identify the points meeting a certain level of performance and leads to a low complexity for searching. In Figure 4, these stored results for each application (App_1 to App_m) are represented as Prf, EC_{CPU}, EC_{GPU}, EC_{MEM}, N_p at various n_b, f_b, n_L, f_L, f_g. In case the architecture is large, the number of design points and their storage overhead might become huge. In such cases, regression model can be derived by using some of the design points and rest can be achieved by using the model [9, 25]. The storage overhead to store the model will be low, but results will not be as accurate as that of storing the design points.

ALGORITHM 1: Run-time Thread-to-core Mapping and Repartitioning of Threads

Input: CNT_{Apps}, $Apps_{Prfr}$, and DPS
Output: TCM and $N_{p'}$ for each application
1: **for** each application A_m **do**
2: Choose points DP_{A_m} ($\in DPS$) such that $Prf > Am_{Prfr}$;
3: **end for**
4: **for** each combination point CP (from CNT_{Apps}) such that total used cores is less than available cores and individual frequencies of big, LITTLE and GPU cores are the same **do**
5: **for** each application in the CP **do**
6: Find stretched execution on GPU core by Equation (10);
7: Find repartitioned work-groups $N_{p'}$ by Equation (15);
8: Find energy consumption $EC^{N_{p'}}$ by Equation (16);
9: Find execution time $ET^{N_{p'}}$ by Equation (19);
10: **end for**
11: Compute energy consumption of CP EC_CP (Equation 20);
12: Add CP with its EC_CP and individual applications $N_{p'}$ and $ET^{N_{p'}}$ in set CPS;
13: **end for**
14: Select the combination point having minimum energy consumption ($minEC\_CP$) and satisfying $Apps_{Prfr}$;
15: For the $minEC\_CP$, return individual applications repartition $N_{p'}$ and number of used cores, their types and frequencies as TCM;

5.3 Run-Time Mapping and Repartitioning for Concurrent Execution of Applications

For a set of concurrent applications to be executed at run-time, this step identifies energy efficient mapping of applications' threads to cores and repartitioning of threads of each application by taking the design points of individual application and applications extended execution time due to co-scheduling into account. The repartitioning process needs to ensure that performance requirement of each application is satisfied and the total number of used MPSoC cores should not exceed than that of available ones.

Algorithm 1 describes the run-time algorithm to perform thread-to-core mapping and repartitioning of threads for each application. The algorithm takes concurrent applications (CNT_{Apps}), their performance requirements ($apps_{Prfr}$) and design points (generated in the previous step as set of design points $DPS = \{D_1, \ldots, D_p, \ldots, D_{max}\}$) as input and provides the following output for each application: *i)* thread-to-core mapping (TCM) in terms of number of used cores, their types and operating frequencies, and *ii)* repartitioned threads ($N_{p'}$) representing the fraction of work-groups to be executed on CPU.

For each application, the algorithm first chooses performance requirement satisfying points from its storage design space, e.g. Figure 6. Then, for each combination point CP (formed by considering one point from each application), if the total number of used CPU cores is less than or equal to the number of available CPU cores and individual frequencies of used big, LITLLE and GPU cores are the same, each application's (App_m) stretched execution time due to co-scheduling, repartitioned workload (work-groups) $N_{p'}$ that defines fraction of work-groups to run on CPU cores, total energy consumption ($EC^{N_{p'}}$) and execution time ($ET^{N_{p'}}$) based on the new partition $N_{p'}$ are computed. Since all the cores within each cluster (big, LITTLE, or GPU) operate at the same frequency, only relevant combination points are evaluated to reduce the run-time overhead. It should be noted that applications' execution is not stretched on CPU cores as spatially isolated

cores are chosen for each application and thus execution time on the CPU side remains almost the same as earlier. However, each application accesses the GPU cores one after another as sequential en-queuing of threads takes place. Therefore, on GPU cores, the execution of an application is stretched by the time taken to complete earlier enqueued threads. Considering applications App_1 to App_m are enqueued sequentially, the stretched (new) execution time of App_m on the GPU cores ($ET_{GPU_{App_m}}$) is computed as follows.

$$ET_{GPU_{App_m}} = \sum_{a=App_1}^{App_m} ET_{GPU_a} \tag{10}$$

Where, ET_{GPU_a} will be the same as execution time of the application a on the CPU side (ET_{CPU_a}) as balanced execution takes place between CPU and GPU based on the partitioning identified in the previous step. Therefore, considering the design point a_{D_p} of application a, ET_{GPU_a} is computed as follows.

$$ET_{GPU_a} = ET_{CPU_a} = \frac{1}{Prf_{a_{D_p}}} \tag{11}$$

Based on the above stretched timing on the GPU side, repartitioned work-groups $N_{p'}$ to balance the execution between CPU and GPU cores is computed by assuming $N_{p'}$ work-groups on CPU and $(W - N_{p'})$ work-groups on GPU will complete the execution at the same time, i.e.:

$$N_{p'} \times ET_{CPU_{owg}} = (W - N_{p'}) \times ET_{GPU_{owg}} \tag{12}$$

Where, $ET_{CPU_{owg}}$ and $ET_{GPU_{owg}}$ are the time required to process one work group (owg) on the CPU and GPU cores, respectively, which are computed as follows.

$$ET_{CPU_{owg}} = \frac{ET_{CPU_{App_m}}}{N_p} \tag{13}$$

$$ET_{GPU_{owg}} = \frac{ET_{GPU_{App_m}}}{W - N_p} \tag{14}$$

Where, $ET_{CPU_{App_m}}$ is the execution time of the application on CPU side and will remain the same as isolated execution is performed. Therefore, it can be computed by using Equation (11). $ET_{GPU_{App_m}}$ can be employed from Equation (10). N_p is earlier partition obtained from the design point.

In order to find the repartition $N_{p'}$, Equation (12) can be derived as:

$$N_{p'} = \frac{W}{1 + \frac{ET_{CPU_{owg}}}{ET_{GPU_{owg}}}} \tag{15}$$

Similarly, repartition is obtained for each application so that balanced execution can be performed for each of them.

The energy consumption for each application ($EC^{N_{p'}}$) based on the repartition $N_{p'}$ is computed as follows.

$$EC^{N_{p'}} = N_{p'} \times EC_{CPU_{owg}} + (W - N_{p'}) \times EC_{GPU_{owg}} + EC_{MEM} \tag{16}$$

Where, $EC_{CPU_{owg}}$ and $EC_{GPU_{owg}}$ are the energy consumption to process one work group (owg) on the CPU and GPU cores, respectively, which are computed as follows.

$$EC_{CPU_{owg}} = \frac{EC_{CPU}}{N_p} \tag{17}$$

$$EC_{GPU_{owg}} = \frac{EC_{GPU}}{W - N_p} \tag{18}$$

EC_{CPU} and EC_{GPU} are the energy consumptions on CPU and GPU cores, respectively, which can be obtained from the design point D_p of the application. The earlier partition N_p can also be obtained from D_p. In Equation (16), it is assumed that EC_{MEM} will be constant as the same amount of data needs to reside in the memory whether they are processed on CPU, GPU or both of them.

The new execution time ($ET^{N_{p'}}$) for each application based on the repartition $N_{p'}$ is computed as follows.

$$ET^{N_{p'}} = \max\{N_{p'} \times ET_{CPU_{owg}}, (W - N_{p'}) \times ET_{GPU_{owg}}\} + \delta \tag{19}$$

Where, δ is memory contention overhead (data transfer delays) due to concurrent execution. The contention overhead is generated due to memory interference among the applications and evaluated in Section 6.2.2.

The total energy consumption for all the concurrent applications within a combination point is found by adding energy consumption of individual applications.

$$EC\_CP = \sum_{\forall CNT_{Apps}} EC^{N_{p'}} \tag{20}$$

After above computations for different combination points, each combination point CP with its energy consumption EC_CP and individual applications repartition $N_{p'}$ and execution time $ET^{N_{p'}}$ are added to a set CPS. Then, the combination point having minimum energy consumption $minEC\_CP$ and satisfying the performance requirement of each application ($1/ET^{N_{p'}} < App_{Perfr}$) is chosen. For this chosen point, the number of used cores, their types and operating frequencies for each application are returned as the thread-to-core mapping TCM and $N_{p'}$ as the repartition. Our approach is generic, but one time profiling is required when the application or platform changes. In case a new application needs to be executed and its profiling results are not available, the best effort [33] or online learning heuristics [32] can be employed to obtain the mapping and repartition, but achieved results might not be efficient.

Each application is executed on the heterogeneous MPSoC by following the TCM and $N_{p'}$. The TCM is controlled by $sched\_setaffinity$ interface in the Linux scheduler. The partitioned application is subsequently executed by enqueuing kernels on the CPU and GPU devices by following the repartition $N_{p'}$.

In case all the applications are not released at the same time, i.e. some are running and other occur, it can be repartitioned based on the available resources and current status of the existing applications, computed as the remaining time to complete them. If existing applications are going to complete soon, the freed resources by them can be considered to decide the repartition of the occurred application, otherwise it should be decided based on the current available resources. This also avoids the overhead of data transfer for existing applications as their mapping and partitioning is not disturbed.

6 EXPERIMENTAL RESULTS

The proposed run-time mapping and threads partitioning approach for energy optimization is extensively evaluated on an Odroid-XU3 platform that runs a modified Ubuntu Linux Kernel 3.10.96. The platform contains Samsung Exynos 5422 heterogeneous MPSoC [5]. The details of the platform are provided in Section 4.1. The proposed approach runs on one of the big (A15) CPU cores. To measure power consumption, the MPSoC also contains four real time current/voltage sensors for four separate power domains: big (A15) CPU cores, LITTLE (A7) CPU cores, GPU cores and DRAM. A power measurement circuit estimates the power as the product of voltage and current, i.e. power = voltage × current. The energy consumption is measured as the product of average power consumption and execution time. Since power is considered for all the domains, the energy

Table 1. Selected Applications from Polybench [16] and SLAMBench [26] Benchmark, Their Number of Work-groups and Performance Requirement (Perf_req (1/Seconds))

Benchmark	App Name	Abbreviation	# work-groups	Perf_req
Polybench	CORRELATION	CR	2048	0.010
	SYR2K	S2	512	0.010
	SYRK	SR	512	0.015
	COVARIANCE	CV	2048	0.011
	2MM	2M	128	0.300
	2DCONV	2D	2048	0.200
	GEMM	GE	512	0.090
	MVT	MV	4096	0.050
SLAMBench	SLAMBench	SB	65536	0.005

consumption of all the software components (e.g., proposed algorithm (Algorithm 1), profiled data, OS, drivers, applications, etc.) running within the chip are included.

The evaluation considers a number of applications from the Polybench benchmark suite [16] and SLAMBench [26]. More details about the benchmark suite are provided in Section 4. To evaluate the applicability of our approach to real-world application, we have considered SLAMBench, which is a computer vision algorithm for 3D scene understanding and solves computationally intensive problem of simultaneous localisation and mapping (SLAM) [26]. Table 1 lists the considered applications and their abbreviations used throughout the paper. Some applications, e.g., 2MM and MVT, have more than one kernel without any dependency between them. In case of dependencies, dataflow between kernels needs to be considered. Further, some applications are memory-bound and some are compute-bound (2MM, GEMM and MVT) [14]. These applications exhibit data parallelism and are written in OpenCL. Since these applications do not have any performance constraint (requirement) in the original benchmark suite, we specify an individual performance requirement for each of them. The constraints are determined such that they are not very tight or loose. This has helped us to map/execute more than one application on the considered MPSoC while satisfying their performance constraints. It should be noted that the performance constraints can be relaxed (made loose) if more applications need to be mapped while satisfying their constraints. They can be made tight as well, but it will lead to performance satisfying mapping/execution of lower number of applications. The applications are considered in various mixes (combinations) randomly to represent a broad spectrum of run-time scenarios.

The proposed approach has been compared against the approaches of [37] and [30], to show energy savings while satisfying performance requirements of concurrent applications. The approach in [37] maps the applications either on CPU or GPU based on the speed-up, which is defined as the time taken on only CPU over only GPU. This approach is referred to as speed-up aware mapping SAM and a comparison of our approach with it shows the potential of mapping and partitioning application workloads on both CPU and GPU. In SAM, if time taken on the GPU is smaller than that of CPU, the application is allocated onto the GPU, otherwise onto the CPU. Therefore, no partitioning is required between CPU and GPU. However, this might end up mapping all the concurrent applications on the GPU or CPU. Therefore, to make a fair comparison, the approach has been modified to map an application on CPU or GPU such that it leads to low overall execution time by distributing applications between CPU and GPU. The approach of [30], follows the same mapping and partitioning that is achieved by considering individual applications and has been referred to as individual application analysis based mapping and partitioning IAAMP. Further, these

Table 2. Mapping (used CPU and GPU Cores and Their Operating Frequencies) and Repartition by Our Approach When Applied to Different Application Scenarios (Single, Double and Triple Concurrent Applications). All the GPU Cores are used for Each Application. Operating Frequencies are not Shown Due to Space Limitation

App scenario	Used CPU cores	Repartition $N_{p'}$
S2-CR	2L+1B : 2L+3B	335: 1634
2D-GE	0L+3B : 0L+1B	140 : 61
SR-S2-CR	1L+1B : 1L+1B : 2L+2B	188: 275: 1740
SR-S2-CV	1L+1B : 1L+1B : 2L+2B	188 : 369 : 1786
GE-2M-2D	1L+1B : 2L+1B : 1L+2B	41: 122 : 681
2M-MV-2D	1L+1B : 1L+1B : 2L+2B	79 : 2380 : 266
GE-MV-2D	0L+1B : 0L+1B : 0L+2B	41 : 3559 : 1337
CR	3L+4B	1695
S2	4L+0B	324
CV	4L+0B	1310
2M	4L+4B	118
2D	2L+4B	316
GE	0L+4B	42
MV	0L+4B	1262

approaches do not consider performance constraints for applications, and thus the constraints are imposed to them in order to make a fair comparison.

Our approach performs energy efficient mapping and partitioning (EEMP) of applications' threads and has been referred to as EEMP. Our approach also finds appropriate voltage/frequency of used cores by employing DVFS. Dynamic power management (DPM) for our approach implies running at the highest voltage/frequency and then shutting down the used cores. Since we perform exploration for all the voltage/frequency points (highest to lowest) and energy consumption is computed only for the active duration of the applications, the approach will suggest using highest voltage/frequency in case DPM is going to lead to lower energy consumption. However, for the considered applications and mixes, our approach identifies some intermediate (between highest and lowest) voltage/frequency points that provide minimum energy consumption while satisfying the performance requirements.

In order to show the effectiveness of proposed approach for various run-time scenarios, the applications are considered in various combinations and individually as well. The first column of Table 2 shows considered run-time scenarios. Based on evaluations (in Section 6.2.2), the contention overhead δ in Equation (19) due to concurrent execution is considered for the worst-case contention, which 3.80% of the performance. Additionally, the performance achieved, run-time overhead and estimation errors for energy consumption and execution time is also evaluated.

6.1 Energy Savings

At a given moment of time, the number of concurrent applications contending for the MPSoC resources may vary. Such scenarios can be observed in a mobile phone where user tries to run more applications at the same time, e.g., internet browser and mp3 player.

For two concurrent applications, a set of two applications from Table 1 are considered to evaluate various approaches for minimizing the energy consumption while meeting performance

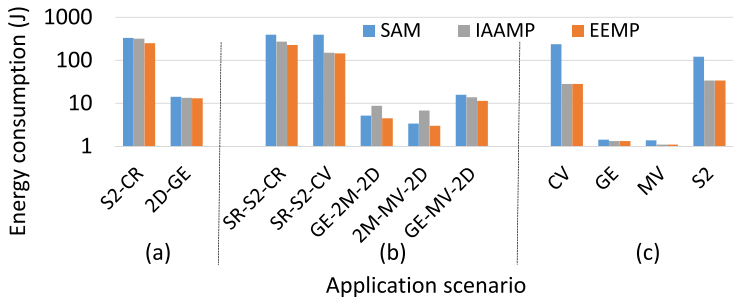

Fig. 7. Energy consumption by employing various approaches for different application scenarios representing (a) 2 concurrent applications, (b) 3 concurrent applications and (c) single application.

requirement of each application. This considered set is shown in the first row and first column of Table 2. Moreover, Table 2 also shows the mapping (in the second column) and repartition (in the third column) for each application when our approach is employed. Figure 7(a) provides the energy consumption results when various approaches are applied. On an average, our approach EEMP achieves 24% energy savings while meeting performance requirements compared to SAM.

To further evaluate the ability of the proposed approach to adapt to execution of concurrent applications, three application scenario, i.e. three concurrent applications are considered. The considered three application scenarios are shown in the second row and first column of Table 2 along with the mapping and repartition of each application obtained by our approach. Figure 7(b) presents total energy consumption values when various approaches are employed. Increase in number of concurrent applications leads to reduced solution space for choosing an energy efficient thread-to-core mapping. This is caused by the resource constraints (see Table 2 for resource combination) and increased contention due to concurrent workloads and demand for meeting their requirements. It has also been observed that IAAMP does not satisfy performance requirements in some scenarios, e.g. 2M-MV-2D. On an average, proposed technique EEMP achieves 28% energy savings while meeting performance requirements compared to SAM.

We also have evaluated single application scenarios. Figure 7(c) shows energy consumption when various approaches are employed. The experimental observation shows that, for most applications our approach provides an efficient repartition. The used CPU cores and obtained repartition for each application can be seen in single application scenario (third row) of Table 2. From Figure 7(c), it can be observed that IAAMP and EEMP outperform SAM. It can be seen the IAAMP and EEMP lead to the same results for single application scenario as the mapping and partitioning of single application's threads is found. On an average, EEMP achieves 46% energy savings while meeting performance requirements compared to SAM.

The four and more applications scenario seems to be not feasible because of high resource contention, leading to not meeting given requirements. Further, even in two and three concurrent application scenarios, the applications having huge memory requirement were not able to run concurrently due limited shared memory on the GPU. For example, 'GPU memory error' has been encountered when CR and CV were tried to run concurrently.

For all evaluated scenarios, on an average, the proposed approach achieves energy savings of 32% compared to existing technique.

6.2 Performance

6.2.1 Performance Deviation. In order to validate the adaptability of the proposed approach to the performance requirements, the achieved performance is compared against the given

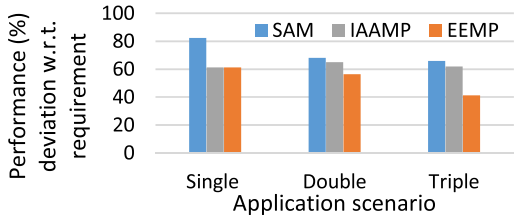

Fig. 8. Performance deviation from the requirement.

performance requirement. Figure 8 shows performance deviation with respect to (w.r.t.) performance requirement when various approaches are employed for different application scenario. The performance deviation is computed as difference between the achieved and required performance. It can be observed that the performance achieved by our approach deviates less than that of SAM and IAAMP. A less deviation by our approach indicates that execution is performed at lower frequencies, which leads to energy savings. Additionally, it can be observed that the deviation by all approaches decreases from single to triple application scenario as applications' execution is stretched due to resource contention caused by limited MPSoC resources.

6.2.2 Effect of Memory Contention. The achieved performance includes the memory interference among the applications as it is computed from the execution time (ET) captured by the system that is measured by considering several factors, e.g. contention and used CPU/GPU cores and their respective frequencies. We also evaluated contention effect as the difference between applications performance when run individually and concurrently. It has been observed that memory contention affects the performance from 1.08% to 3.80% for the considered run-time scenarios.

6.3 Memory Overhead

The profiled data (design points) for each application is stored after applying the storage optimization described in Section 5.2, where each design point is represented by a 10-tuple:
$D_p = (N_p, n_b, f_b, n_L, f_L, f_g, Prf, EC_{CPU}, EC_{GPU}, EC_{MEM})$. The storage overhead for each application before and after the optimization is 23.5 kB and 10.5 kB, respectively. This represents a very low overhead.

6.4 Run-Time Overhead

The run-time overhead of our proposed approach (Algorithm 1) depends on the number of combination points considered for the concurrent applications and time taken to perform various computations (e.g., stretched execution, repartitioned work-groups, energy consumption and execution time) for each application in a combination point. Since computations are done fairly quickly (in the order of μs), the overhead is dominated by the number of combination points. However, since the approach performs computations only for the combination points using less number of cores than available ones and having individual frequencies of big, LITTLE and GPU cores the same, a lot of combination points become redundant. Thus, computations are performed for a small number of combination points, which leads to light weight run-time management. By taking the average over various application scenarios, our run-time approach (Algorithm 1) shows an average overhead of approximately 20 ms.

We also have computed the overhead with respect to the total execution time. Figure 9 illustrates the total run-time overhead, computed as percentage of total execution time, for six application scenarios. The run-time overhead for application scenario *CR-S2-SR*, having a long execution time

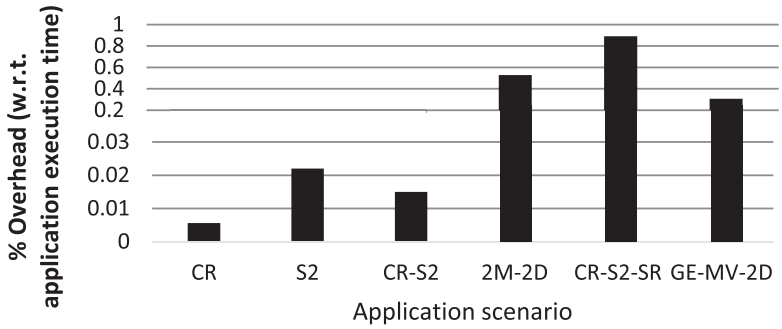

Fig. 9. Run-time overhead of the proposed approach.

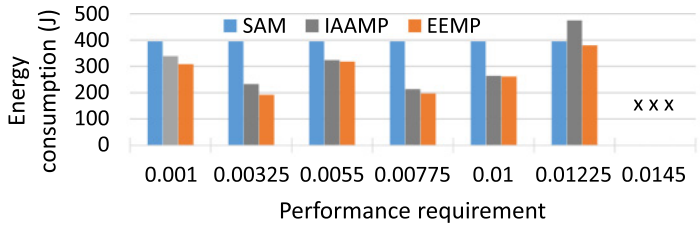

Fig. 10. Energy consumption at varying performance constraint.

of 77 sec is ~0.9%. The average run-time overhead is 0.29%, which is very minimal. This implies that the proposed run-time approach can be efficiently used to find mapping and partitioning of concurrent applications' threads.

6.5 Run-Time Estimation Errors

The proposed run-time algorithm estimates energy consumption by Equation (20). This needs to be estimated as the algorithm needs to know energy consumption of several mapping and partition options quickly in order to identify the design point having minimum energy consumption. Since this is used to decide the final design point, it needs to be estimated accurately. We have compared the estimated energy consumption by Equation (20) with the one achieved after completing the execution and computed by using available MPSoC power sensors. The compared energy consumption results for all the application scenarios have shown that the difference in estimated and observed energy consumption is less than 6%, which provides sufficient accuracy to find minimum energy consumption point.

Similarly, execution time estimated by Equation (19) for each application is compared against the execution time achieved after completing it. This estimation is required as we need to check it against the timing requirements so that only performance satisfying design options are considered. The compared results for various application scenarios have shown that the difference between estimated and observed execution time varies from 3% to 8%, which has helped us to find performance satisfying points with reasonable accuracy.

6.6 Effect of Varying Performance Constraints

We also analyse the effect of varying performance constraints on energy consumption when employing our and existing approaches. Figure 10 shows energy consumption results when performance constraints of concurrent applications CR, S2 and SR are varied from 0.001 to 0.0145. With

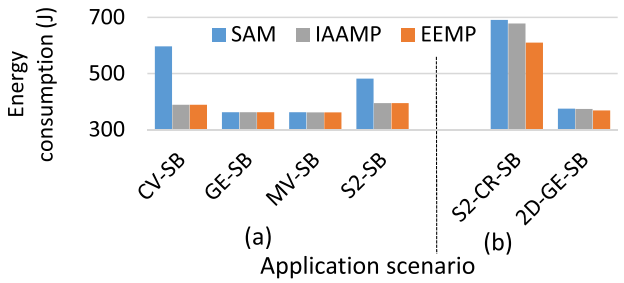

Fig. 11. Energy consumption for real-world application mixes.

tighter (higher) performance constraints, e.g. 0.0145, all the approaches fail to satisfy them (x x x in Figure 10). It can be observed that our approach always provides energy savings over existing approaches.

6.7 Case-study with Real-world Application

Figure 11 shows energy consumption results when various approaches are applied to different application mixes from Ploybench and SLAMBench (SB) benchmark. The SB is run either on the CPU or GPU. Two and three applications are mixed in Figure 11(a) and (b), respectively. For two application mixes, IAAMP and EEMP achieve the same results as SB is mapped on GPU and other is partitioned between CPU and GPU in exactly the same manner by IAAMP and EEMP. For higher number of application mixes, EEMP reduces energy consumption when compared to other approaches.

7 CONCLUSIONS

We proposed a run-time management approach that performs energy efficient mapping and threads partitioning of concurrent applications on CPU-GPU cores of heterogeneous MPSoC. The approach utilizes the knowledge from design-time profiling to identify the mapping in terms of number of used cores, their type and operating frequencies. The profiling knowledge is also used to identify workload distribution between CPU and/or GPU cores. Validation on Odroid-XU3 platform for various application scenarios has shown that our approach can be employed to achieve higher energy savings compared to existing approaches. Towards the development of future energy efficient and feature rich embedded systems with heterogeneous MPSoCs containing CPU and GPU cores, the advances reported in this paper are important contributions. In future, we plan to consider distributed memory architectures requiring movement of data between different devices, e.g. CPU and GPU.

ACKNOWLEDGMENTS

This work was supported in parts by the EPSRC Grant EP/L000563/1 and the PRiME Programme Grant EP/K034448/1 (www.prime-project.org). Experimental data used in this paper can be found at DOI: https://doi.org/10.5258/SOTON/D0164.

REFERENCES

[1] 2013. ARM Mali T628. http://www.arm.com/. (2013).
[2] 2014. ARM big.LITTLE Technology. http://www.arm.com/. (2014).
[3] 2015. Qualcomm Adreno 530 and 540. https://www.qualcomm.com/. (2015).
[4] 2016. ARM Mali 71. http://www.arm.com/. (2016).

[5] 2016. Exynos 5 Octa (5422). www.samsung.com/exynos/. (2016).
[6] 2016. Odroid-XU3. http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127. (2016).
[7] 2016. The open standard for parallel programming of heterogeneous systems. https://goo.gl/A9wXRJ. (2016).
[8] 2017. FreeOCL: Multi-platform implementation of OpenCL 1.2 targeting CPUs. (2017). https://github.com/zuzuf/freeocl
[9] Ali Aalsaud, Rishad Shafik, Ashur Rafiev, Fie Xia, Sheng Yang, and Alex Yakovlev. 2016. Power–Aware Performance Adaptation of Concurrent Applications in Heterogeneous Many-Core Systems. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 368–373.
[10] Karunakar Reddy Basireddy, Amit Kumar Singh, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2017. ITMD: run-time management of concurrent multi-threaded applications on heterogeneous multi-cores. In *Conference on Design, Automation and Test in Europe (DATE), University Booth*. 1.
[11] Kiran Chandramohan and Michael F. P. O'Boyle. 2014. Partitioning data-parallel programs for heterogeneous MP-SoCs: time and energy design space exploration. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 73–82.
[12] E. Del Sozzo, G. C. Durelli, E. M. G. Trainiti, A. Miele, M. D. Santambrogio, and C. Bolchini. 2016. Workload-aware power optimization strategy for asymmetric multiprocessors. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 531–534.
[13] Bryan Donyanavard, Tiago Mück, Santanu Sarma, and Nikil Dutt. 2016. SPARTA: runtime task allocation for energy efficient heterogeneous many-cores. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 27.
[14] L. Bagnères et al.Switchable scheduling for runtime adaptation of optimization. In *Euro-Par'14*. 222–233.
[15] Ivan Grasso, Petar Radojkovic, Nikola Rajovic, Isaac Gelado, and Alex Ramirez. 2014. Energy efficient hpc on embedded socs: Optimization techniques for mali gpu. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 123–132.
[16] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–10.
[17] Peter Greenhalgh. 2011. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper* (2011), 1–8.
[18] Dominik Grewe and Michael F. P. O'Boyle. 2011. A static task partitioning approach for heterogeneous systems using OpenCL. In *International Conference on Compiler Construction*. Springer, 286–305.
[19] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. 2013. OpenCL task partitioning in the presence of GPU contention. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 87–101.
[20] Timo Hönig, Heiko Janker, Christopher Eibel, Oliver Mihelic, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. 2014. Proactive Energy-Aware Programming with PEEK. In *TRIOS*.
[21] Gangwon Jo, Won Jong Jeon, Wookeun Jung, Gordon Taft, and Jaejin Lee. 2014. OpenCL framework for ARM processors with NEON support. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM, 33–40.
[22] Ali Karami, Farshad Khunjush, and Seyyed Ali Mirsoleimani. 2015. A statistical performance analyzer framework for OpenCL kernels on Nvidia GPUs. *The Journal of Supercomputing* 71, 8 (2015), 2900–2921.
[23] David H. K. Kim, Connor Imes, and Henry Hoffmann. 2015. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2015 IEEE 3rd International Conference on*. IEEE, 78–85.
[24] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 45–55.
[25] Jun Ma, Guihai Yan, Yinhe Han, and Xiaowei Li. 2016. An Analytical Framework for Estimating Scale-Out and Scale-Up Power Efficiency of Heterogeneous Manycores. *IEEE Trans. Comput.* 65, 2 (2016), 367–381.
[26] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, and others. 2015. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE, 5783–5790.
[27] Prasanna Pandit and R. Govindarajan. 2014. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 273.
[28] Indrani Paul, Vignesh Ravi, Srilatha Manne, Manish Arora, and Sudhakar Yalamanchili. 2014. Coordinated energy management in heterogeneous processors. *Scientific Programming* 22, 2 (2014), 93–108.
[29] Behnaz Pourmohseni, Michael Glaß, and Jürgen Teich. 2017. Automatic operating point distillation for hybrid mapping methodologies. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1135–1140.

[30] Alok Prakash, Siqi Wang, Alexandru Eugen Irimiea, and Tulika Mitra. 2015. Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms. In *IEEE International Conference on Computer Design (ICCD)*. IEEE, 208–215.

[31] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. 2017. A Survey and Comparative Study of Hard and Soft Real-Time Dynamic Resource Allocation Strategies for Multi-/Many-Core Systems. *ACM Comput. Surv.* 50, 2, Article 24 (2017), 40 pages.

[32] Amit Kumar Singh, Charles Leech, Karunakar Reddy Basireddy, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2017. Learning-based Run-time Power and Energy Management of Multi/Many-core Systems: Current and Future Trends. In *Journal of Low Power Electronics (JOLPE)*. 26.

[33] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *Proceedings of the Design Automation Conference (DAC)*. Article 1, 10 pages.

[34] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 213–224.

[35] Hao Wang, Vijay Sathish, Ripudaman Singh, Michael J. Schulte, and Nam Sung Kim. 2012. Workload and power budget partitioning for single-chip heterogeneous processors. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 401–410.

[36] Hao Wang, Ripudaman Singh, Michael J. Schulte, and Nam Sung Kim. 2014. Memory scheduling towards high-throughput cooperative heterogeneous computing. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 331–342.

[37] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 1–10.

[38] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. 2015. VirtCL: a framework for OpenCL device abstraction and management. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 161–172.

Received April 2017; revised June 2017; accepted June 2017